Die Programmiersprache Modula-P

Jürgen Vollmer
Gesellschaft für Mathematik und Datenverarbeitung
Forschungsstelle an der Universität Karlsruhe

Zusammenfassung

Die Programmiersprache *Modula-P* erweitert *Modula-2* um parallele Sprachkonstrukte. Sie basiert auf den *kommunizierenden sequentiellen Prozessen* (CSP) und den bedingten kritischen Regionen.

Dieser Artikel erweitert die frühere Sprachdefinition von *Modula-P* und definiert die dritte Version der Sprache. Einige typische Programmbeispiele werden gezeigt.

1 Einführung

Heutzutage sind die verschiedenartigsten parallelen Rechner am Markt erhältlich, von deren Nutzung man sich höhere Rechenleistungen verspricht. Es stellt sich nun die Frage, wie diese Rechner zu programmieren sind. Es gibt hierfür zumindest zwei Ansätze: Entweder kann der Übersetzer die in einem sequentiellen Programm enthaltene Parallelität aufspüren und versuchen, dieses automatisch zu parallelisieren. Oder der Programierer[1] gibt die dem *Problem* inhärente Parallelität explizit im Programm an. Mit *Modula-P* wird der zweite Weg beschritten.

Modula-P [4, 5] ist eine Obermenge der Sprache *Modula-2* [6], erweitert um Sprachkonstrukte, die auf den *kommunizierenden sequentiellen Prozessen (CSP)* [2] und bedingten kritischen Regionen [1] basieren.

Die *Modula-P* zugrundeliegende Rechnerarchitektur ist der sogenannten *MIMD*[2]-Rechner mit gemeinsamem und/oder getrenntem Speicher. Eine solche Architektur besteht aus mehreren Prozessoren, die alle üblichen arithmetischen und logischen Operationen ausführen können. Sie kommunizieren miteinander entweder über einen gemeinsamen Speicher oder über spezielle Hardware (oder beides).

Diese Sprachdefinition ergänzt die bisherigen Sprachdefinitionen [4, 5]. Neben einigen syntaktischen Änderungen wird das Konzept der *bedingten kritischen Regionen* eingeführt.

Es gibt für *Modula-P* eine Entwicklungsumgebung *MOCKA-P* (Modula-P Compiler Karlsruhe), die Code für ein Netzwerk von T800 Transputern [3] erzeugt. Geplant und in Entwicklung sind Portierungen auf T9000 Prozessoren, sowie auf einen Mehrprozessorrechner mit gemeinsamen Speicher.

Die nächsten Abschnitte definieren die Sprache und stellen Lösungen für einige typische Probleme dar.

2 Die Sprache

Alle aus *Modula-2* [6] bekannten Sprachkonstrukte (mit Ausnahme der Koroutinen) sind Teil von *Modula-P*.

Modula-P erlaubt es, Anweisungen parallel (`PAR` Anweisung) auf einem oder mehreren Prozessoren (`AT` Anweisung) auszuführen. Diese Platzierung kann „netzwerkunabhängig“

*Diese Arbeit wurde teilweise durch das ESPRIT Projekt COMPARE (#5399) gefördert.

[1] *Programmierer* steht für beide: den Programmierer und die Programmiererin.

[2] Multiple Instruction Multiple Data

gestaltet werden, indem relative Positionen wie `north, east` etc. benutzt werden. Die Zahl der kreierten Prozesse kann zur Laufzeit bestimmen werden.
Prozesse können auf zwei Arten miteinander kommunizieren: sie können auf gemeinsame Variablen zugreifen, oder aber sich gegenseitig Nachrichten schicken. Der Zugriff auf gemeinsame Variablen unterscheidet sich nicht vom Zugriff auf nicht-gemeinsame Variablen. Nachrichten werden über Kanäle (Variable vom Typ `CHANNEL OF ...`) mit der Sende- (`!`) und Empfangeanweisung (`?`) synchron ausgetauscht. Die Nachrichten werden automatisch vom Sender zum Empfänger durch das Kommunikationsnetzwerk des Rechners geleitet, ohne daß der Programmierer den Weg der Nachricht beschreiben muß.
Die Prozesse, die gemeinsame Variablen als Kommunikationsmedium wählen, können allerdings nur dann auf verschiedenen Prozessoren abgearbeitet werden, wenn diese Prozessoren auch wirklich auf einen gemeinsamen Speicher zugreifen können. Ansonsten werden diese Prozesse auf einem Prozessor abgearbeitet. Prozesse, die nur mittels Nachrichten kommunizieren, können dagegen immer verteilt ausgeführt werden.
Ein Prozeß kann simultan auf mehrere Ereignisse warten (`ALT` Anweisung). Bedingte kritische Regionen werden durch die `LOCK` und `TRY` Anweisung und einen neuen Variablentyp `GATE` unterstützt.
Eine weitere Art von Übersetzungseinheit, *Unit*, wird eingeführt. Sie definiert gleichzeitig neue „Lebensdauer“ von Variablen.
Die Syntax für die Erweiterung der Anweisungen lautet[3]:

```
statement ::= ... | ParStatement | AtStatement | ChannelStatement |
              CCRStatement | AltStatement .
```

2.1 Parallele Ausführung

Die Ausführung einer Anweisungsfolge wird *Prozeß* genannt. `PAR` p_1 | $\cdots$ | p_n `END` spezifiziert die nebenläufige Ausführung ihrer Komponenten $p_1, \cdots, p_n$. Die erzeugten Prozesse werden *Kindprozesse* des Prozesses genannt, der diese `PAR` Anweisung ausführt, d.h. des *Vaterprozesses*.
Beim Ausführen einer `PAR` Anweisung wird der Vaterprozeß so lange suspendiert, bis alle Kindprozesse terminiert haben. Ein Prozeß terminiert, wenn die letzte Anweisung seiner Anweisungsfolge seine Ausführung beendet hat. Der Anweisungsfolge der Komponente p_i kann ein *Replikator* vorausgehen. Ein Replikator erzeugt mehrere Prozesse, die alle die gleichen Anweisungen ausführen (siehe Abschnitt 2.7).
`EXIT` Anweisungen, die nicht zu in `PAR` enthaltenen Schleifen gehören, und `RETURN` Anweisungen sind nicht als Anweisungen in der Anweisungsfolge von p_i erlaubt.
Die Syntax lautet:

```
ParStatement ::= PAR Process {"|" Process} END .
Process      ::= [[replicator] StatementSequence] .
```

2.2 Verteilte Ausführung

Die Syntax der `AT` Anweisung lautet:

```
AtStatement ::= AT Expression DO StatementSequence END .
```

Die `AT` Anweisung erlaubt es, die in ihr enthaltenen Anweisungen auf einem anderen Prozessor aufführen zu lassen[4]. Die Bedeutung eines Programms ist unabhängig davon, ob die Ausführung von Anweisungen durch `AT` auf einem anderen Prozessor erfolgt oder nicht. Eine Ausnahme bilden solche Unterschiede, die sich durch möglicherweise entstehende verschiedene Ausführungszeiten ergeben.

[3] Die Syntax wird an die in [6] angegebene angelehnt.
[4] Dies ist eine Erweiterung des *Remote Procedure Call* Konzeptes.

Der Ausdruck `Expression` muß zuweisungskompatibel zum Typ `INTEGER` sein. Die beabsichtigte Bedeutung des Wertes von `Expression` ist es, einen Prozessor zu spezifizieren, der die Anweisungen `StatementSequence` ausführen soll. Der Wert von `Expression` wird vom Laufzeitsystem[5] interpretiert. Das Laufzeitsystem definiert eine Menge von erlaubten Werten für `Expression`, sowie das Verhalten, wenn illegale Werte angegeben werden. Das Bibliotheksmodul `Net` bietet eine Reihe von Funktionen[6] an, die es ermöglichen, legale Werte für `Expression` zu bestimmen.
Das Laufzeitsystem kann die Art der Anweisungen einschränken, die auf einem anderen Prozessor ausgeführt werden dürfen. Es ist allerdings kein Fehler, wenn solche Anweisungen in der `AT` Anweisung enthalten sind; die ganze Anweisungsfolge wird dann einfach nicht auf einem anderen Prozessor ausgeführt. Eine typische Einschränkung ist: auf einem Rechnersystem ohne gemeinsamen Speicher können Anweisungen nicht verteilt ausgeführt werden, wenn sie gemeinsame Variablen benutzen.

2.3 Nachrichtenaustauch über Kanäle

Prozesse können über Kanäle synchron Nachrichten austauschen. Synchron bedeutet, daß der Prozeß, der eine Kommunikationsanweisung über Kanal k zuerst erreicht, so lange wartet, bis ein Partnerprozeß seine korrespondierende Kommunikationsanweisung über k ausführt. Dann wird die Nachricht ausgetauscht, und beide Prozesse fahren in der Abarbeitung ihrer Anweisungen unabhängig fort.
Es gibt n:m- und 1:1-Kanäle. Bei n:m Kanälen können n Sender und m Empfänger einen Kanal k benutzen. Zu einem Zeitpunkt tauchen aber nur genau ein Sender und ein Empfänger eine Nachricht über k aus. Sind mehrere Sender und Empfänger bereit, eine Nachricht auszutauschen, werden ein Sender und ein Empfänger aus der Menge der bereiten Prozesse zur Kommunikation ausgewählt. Alle anderen Prozesse warten weiter, bis sie kommunizieren dürfen. Es ist sichergestellt, daß die Auswahl *fair* ist, d.h. jeder Sender und Empfänger kommt irgendwann zum Zug.
Bei 1:1-Kanälen sind nur ein Sender und ein Empfänger erlaubt. Versuchen mehrere Prozesse gleichzeitig zu senden oder zu empfangen, ist dies ein Fehler.
Kanäle werden wie *Modula-2* Variablen behandelt, d.h. sie haben einen Typ (`CHANNEL OF` *MessageType*), müssen deklariert und initialisiert werden. Sie können an Prozeduren als Parameter übergeben und anderen Variablen des gleichen Typs zugewiesen werden. Nachrichten (d.h. Werte vom Typ *MessageType*) können von jedem Typ aus *Modula-P* sein.

```
TYPE MessageType=...; tChannel=CHANNEL OF MessageType; VAR ch:tChannel;
```

deklariert einen Kanaltyp `tChannel` mit dem Nachrichtentyp `MessageType` und eine Kanalvariable `ch`, über den Nachrichten vom Typ `MessageType` geschickt werden können.
Vor der ersten Kommunikation eines Kanales muß dieser einmal mit der Standardprozedur `INIT` initialisiert werden. Die Signatur von `INIT` ist:

```
PROCEDURE INIT (VAR channel : ChannelType; info : BITSET);
```

`channel` ist eine Kanalvariable, `info` ist eine Menge von Informationen, die vom Laufzeitsystem festgelegt und benötigt werden, z.B. ob es sich um einen 1:1- oder n:m-Kanal handelt. Werte für `info` sind in dem Modul `Channel` definiert[7]. Der `info` Parameter ist optional. Fehlt die Angabe von `info`, wird `channel` ein 1:1-Kanal.
Durch `INIT` erhält die Variable ein für alle Prozesse eindeutiges Kennzeichen. Es wird bei der Parameterübergabe, Zuweisung, etc. weitergegeben. Zwei Kanal-Variablen mit dem

[5] Zum *Laufzeitsystem* gehört sowohl das Rechnersystem, auf dem dieses Programm abläuft, als auch das Betriebssystem, das die Ausführung von *Modula-P* Programmen unterstützt.

[6] Zum Beispiel: `north()`, `south()`, etc. die die Nachbarn eines Prozessors kennzeichnen. Diese liefern auch dann legale Nummer, wenn es keinen entsprechenden Nachbarprozessor gibt.

[7] Z.B. die Konstanten: `n2m, one2one`.

gleichen Nachrichtentyp sind gleich, wenn sie das gleiche Kennzeichen tragen. Sie können mit = bzw. # verglichen werden.
Die Anweisung `channel ! expression` sendet den Wert von `expression` über den Kanal `channel`. Die Anweisung `channel ? variable` empfängt eine Nachricht vom Kanal `channel` und weist sie der Variablen `variable` zu. `expression` muß zum Nachrichtentyp von `channel` zuweisungskompatibel sein. Der Nachrichtentyp von `channel` muß zum Typ von `variable` zuweisungskompatibel sein.
Eine Nachricht beliebigen Typs kann über einen Kanal vom Typ `CHANNEL OF ANY`[8] geschickt werden. Dazu müssen die Länge und Adresse der Nachricht angegeben werden:

```
VAR ch : CHANNEL OF ANY; ch ! src_adr, size; ch ? dest_adr, size;
```

`ch` muß vom Typ `CHANNEL OF ANY` sein. `src_adr` bzw. `dest_adr` sind Ausdrücke vom Typ `ADDRESS`. `src_adr` ist die Adresse des Speicherbereiches, in dem die Nachricht steht, die gesendet werden soll. `dest_adr` ist die Adresse des Speicherbereiches, in dem die empfangene Nachricht abgespeichert werden soll. `size` ist ein Ausdruck, der zuweisungskompatibel zu `CARDINAL` sein muß. `size` gibt die Länge der Nachricht in Bytes an. Die Nachrichtenlängen von Sender und Empfänger müssen übereinstimmen. Der Empfängerprozeß muß vor dem Empfangen der Nachricht genügend Speicherplatz für sie bereitstellen. Die Nachrichtenlänge kann von einer zur nächsten Kommunikation variieren.
Funktionen können einen Kanal als Ergebnistyp haben.
Die Syntax lautet:

```
type             ::= ... | ChannelType .
ChannelType      ::= CHANNEL OF MessageType | CHANNEL OF ANY .
MessageType      ::= type.
ChannelStatement ::= SendStatement | ReceiveStatement .
SendStatement    ::= channel "!" expression | channel "!" adr, size .
ReceiveStatement ::= channel "?" designator | channel "?" adr, size .
channel          ::= designator .
adr, size        ::= expression .
```

2.4 Gemeinsame Variablen

Mehrere Prozesse können (gleichzeitig) auf eine gemeinsame Variable lesend und schreibend zugreifen. Gemeinsame Nutzung von Variablen unterscheidet sich syntaktisch nicht von anderer Nutzung. Um die bekannten Probleme bei der Nutzung von gemeinsamen Variablen zu vermeiden, sollten die Anweisungen aus Abschnitt 2.5 benutzt werden.

2.5 Bedingte kritische Regionen

Kritische Regionen werden benötigt, wenn Ressourcen (z.B. Variablen, Dateien, Datenstrukturen, usw.) zu einem Zeitpunkt von nur genau einem Prozeß, oder einer bestimmten maximalen Anzahl von Prozessen gleichzeitig benutzt werden dürfen. Die Anweisungen einer *bedingten* kritischen Regionen können von einem Prozeß nur dann ausgeführt werden, wenn zudem eine bestimmte Bedinung erfüllt ist. Kritische Regionen erlauben eine strukturierte Benutzung von binären bzw. zählenden Semaphoren.

2.5.1 Der Typ GATE

Modula-P kennt einen weiteren skalaren Typ `GATE`, dessen Werte als Bedingung in bedingten kritischen Regionen dienen.

```
PROCEDURE INIT (VAR gate : GATE; max : CARDINAL);
```

Durch Aufruf der Standardprozedur `INIT`, erhält die Variable `gate` einen definierten Zustand. Jedes Gate hat zwei Zähler: `gate.cur` und `gate.max`. Durch `INIT` werden sie zu

[8] Bemerkung: `ANY` ist ein Schlüsselwort und nicht ein eigener Typ.

gate.cur = 0 und gate.max = max bestimmt. Der optionale Parameter max ($\geq$ 1) hat den Defaultwert 1.
Ebenfalls durch INIT erhält gate ein für alle Prozesse systemweit eindeutiges Kennzeichen. Dieses eindeutige Kennzeichen wird bei der Parameterübergabe, Zuweisung, etc. weitergegeben. Zwei Gate-Variablen sind gleich, wenn sie das gleiche Kennzeichen tragen, sie können mit = bzw. # verglichen werden.
Bei undefiniertem Wert einer Gate-Variablen ist das Verhalten von LOCK, TRY und LOCKED undefiniert.

2.5.2 Die Anweisungen LOCK und TRY

Bedingte kritische Regionen werden in *Modula-P* durch die LOCK und TRY Anweisungen ausgedrückt. Die LOCK Alternative der ALT Anweisung (Abschnitt 2.6) wird wie die LOCK Anweisung behandelt. Jeder dieser Anweisungen ist eine Gate-Variablen zugeordnet. Haben die Gate-Variable von verschiedenen LOCK bzw. TRY Anweisungen das gleiche Kennzeichen, bilden sie zusammen die bedingte kritische Region.

```
LOCK gate, nr DO stmts END
TRY gate, nr DO stmts1 ELSE stmts2 END
```

stmts bzw. $\texttt{stmts}_1$ werden die *geschützen Anweisungen* von LOCK bzw. TRY genannt. nr ist ein Ausdruck, der den Typ CARDINAL haben muß. Die Angabe von nr ist optional, der Defaultwert ist 1.
Die geschützen Anweisungen können von einem Prozeß nur ausgeführt werden, wenn gilt:

$$\texttt{nr} + \texttt{gate.cur} \leq \texttt{gate.max} \tag{1}$$

Vor Ausführung der geschützen Anweisungen wird gate.cur um nr inkrementiert, danach wieder um nr dekrementiert.
Mit anderen Worten: es können höchstens gate.max Prozesse die mit gate geschützen Anweisungen ausführen, d.h. die kritische Region betreten. Dabei kann ein Prozeß mehrfach zählen, indem nr > 1 gilt.
Bei einer LOCK Anweisung wird gewartet, bis Bedingung (1) erfüllt ist. Warten mehrere Prozesse auf den Eintritt in eine kritische Region, ist Fairness sichergestellt.
Im Gegensatz dazu wird bei TRY nicht gewartet: ist Bedingung (1) nicht erfüllt, werden sofort die optionalen Anweisungen $\texttt{stmts}_2$ ausgeführt. Fehlen sie, entspricht TRY in diesem Falle der leeren Anweisung.
Die Standardfunktion LOCKED (gate : GATE; nr : CARDINAL) : BOOLEAN liefert TRUE genau dann, wenn Bedingung (1) nicht erfüllt ist. nr ist optional, der Defaultwert ist 1.
Die Syntax lautet:

```
type         ::= ... | GATE .
CCRStatement ::= LOCK gate ["," expression] DO StatementSequence END |
                 TRY gate ["," expression] DO StatementSequence
                     [ELSE StatementSequence] END .
gate         ::= designator .
```

LOCK und TRY Anweisungen können auch geschachtelt sein, es kann jedoch bei geschachtelten LOCKs leicht zu Verklemmungen kommen. Das folgende Beispiel demonstriert dies: Führen die beiden Prozesse die Anweisungen (* 1 *) bzw. (* 2 *) aus, haben sie bereits jeder eine kritsche Region betreten. Die Verklemmung tritt ein, da nun jeder Prozeß die Anweisungen einer zweiten kritischen Region ausführen will, die aber bereits von dem jeweils anderen Prozeß „besetzt“ ist.

```
PAR
   LOCK a DO (* 1 *) LOCK b DO ... END END;
 |
   LOCK b DO (* 2 *) LOCK a DO ... END END;
END
```

2.6 Selektives Warten

Ein Prozeß kann auf mehrere Ereignisse mittels der ALT Anweisung gleichzeitig warten. Tritt dann ein Ereignis ein, werden die entsprechende Anweisungen ausgeführt.

ALT $\texttt{event}_1$:$\texttt{stmts}_1$ | $\texttt{event}_2$:$\texttt{stmts}_2$ | ... | $\texttt{event}_n$:$\texttt{stmts}_n$ ELSE $\texttt{stmts}_0$ END

Der Prozeß, der eine ALT Anweisung ausführt, wird solange suspendiert, bis eines der Ereignisse aus der Menge $\{\texttt{event}_1, \cdots, \texttt{event}_n\}$ eintrifft. Trifft ein Ereignis, oder treffen gleichzeitig mehrere Ereignisse ein, wird aus der Menge der eingetroffenen Ereignisse ein beliebiges $\texttt{event}_i$ ausgewählt und die entsprechenden Anweisungen $\texttt{stmts}_i$ ausgeführt.
Es gibt vier Arten von Ereignissen: a) der Kommunikationswunsch eines anderen Prozesses, b) der Wunsch, eine bedingte kritische Region betreten zu können, c) der Ablauf einer gewissen Zeitspanne, sowie d) das Ereignis, das „immer eintrifft“:

a) `expression , channel ? variable` oder `expression , channel ? adr, size`
b) `expression , LOCK gate , nr`
c) `expression , WAIT (ticks)`
d) `expression`

Der Ausdruck `expression` muß ein boolsches Ergebnis liefern. In den ersten drei Fällen ist er optional und hat dann den Defaultwert `TRUE`. `channel` muß eine Kanalvariable sein, `variable` eine dazu passende Variable, `adr` muß ein Ausdruck vom Typ `ADDRESS`, `size` ein Ausdruck vom Typ `CARDINAL` sein. `gate` ist eine Variable vom Typ `GATE`, `nr` ist ein optionaler `CARDINAL` Ausdruck, dessen Defaultwert `1` ist, und `ticks` ist ein Ausdruck, der einen `INTEGER` Wert liefert.
Ein Ereignis tritt ein, wenn `expression` zu `TRUE` ausgewertet wird. Zusätzlich muß für die Ereignisse a) bis d) die jeweilige Bedingung a) bis d) erfüllt sein:

a) Es muß ein anderer Prozeß zum Senden einer Nachricht über diesen Kanal bereit sein.
b) Für `gate` tritt die Bedingung (1) ein.
c) Die Zeit, die durch `ticks` angegeben wird, ist seit Beginn der Ausführung von `ALT` verstrichen.
d) Dieses Ereignis tritt immer und sofort ein.

Wertet sich keiner der Ausdrücke `expression` der `ALT` Anweisung zu `TRUE` aus, werden die Anweisungen des `ELSE`-Teiles $\texttt{stmt}_0$ ausgeführt. Fehlt der optionale `ELSE` Teil, wird in diesem Falle ein Fehler gemeldet. Wird das Ereignis

a) ausgewählt, findet zuerst der Nachrichtenaustausch statt,
b) ausgewählt, werden die zugehörigen Anweisungen als mit `gate` geschützte Anweisungen betrachtet und ausgeführt,
c) oder **d)** ausgewählt, werden die zugehörigen Anweisungen ausgeführt.

Die Syntax lautet:

```
AltStatement ::= ALT alternative { "|" alternative }
                         [ELSE StatementSequence] END .
alternative  ::= [[replicator] event ":" StatementSequence] .
event        ::= [expression ","] ReceiveStatement |
                 [expression ","] LOCK gate ["," expression] |
                 [expression ","] WAIT "(" ticks ")" |
                 expression .
ticks        ::= expression.
```

2.7 Replikatoren

Die Komponenten der `PAR` und `ALT` Anweisung können repliziert werden.
`[var : lower TO upper] stmts` kann als Komponente einer `PAR` Anweisung auftreten. `(ORD(upper)-ORD(lower)+1)` verschiedene Prozesse führen dann nebenläufig die Anweisungen `stmts` aus. In jedem Prozeß hat die Replikatorvariable `var` einen eindeutigen, von allen anderen Prozessen dieses Replikators verschiedenen Wert aus dem Wertebereich `[lower .. upper]`.

`[var : lower TO upper] event : stmts` kann als Komponente einer `ALT` Anweisung auftreten. Auf `(ORD(upper)-ORD(lower)+1)` Alternativen wird gleichzeitig gewartet. In jeder dieser Alternativen hat die Replikatorvariable `var` einen eindeutigen, von allen anderen Alternativen dieses Replikators verschiedenen Wert aus dem Wertebereich `[lower .. upper]`.
Für `var`, `lower` und `upper` gelten die gleichen Regeln wie in einer `FOR` Anweisung. Gilt `(ORD(upper)-ORD(lower)+1)`$\leq$`0`, wird kein Prozeß, bzw. keine Alternative erzeugt.
Die Syntax lautet:

```
replicator  ::= "[" ident ":" lower TO upper "]" .
lower, upper ::= expression .
```

2.8 INOUT Parameterübergabe

In *Modula-2* gibt es zwei Arten, Parameter an eine Prozedur zu übergeben: entweder als *Wert-* oder *(Var) Referenzparameter*. Letztere werden durch das Schlüsselwort `VAR` in der formalen Parameterliste gekennzeichnet.
Modula-P kennt noch eine dritte Möglichkeit der Parameterübergabe: per *Wert und Resultat* (Inout-Parameter). Sie wird durch das Schlüsselwort `INOUT` gekennzeichnet. Es gelten die gleichen Regeln wie für Var-Parameter, insbesondere sind also nur Variablen als aktuelle Parameter erlaubt.
Inout-Parameter werden übergeben, indem der Wert des aktuellen Parameters der Variablen zugewiesen wird, für die der formale Parameter steht. Nach Ende der Prozedur, wird der Wert des formalen Parameters an den aktuellen Parameter (Variable) zugewiesen.
Die Syntax für die formalen Parameter in Prozedurdeklarationen und -typen lautet:

```
FPSection     ::= [pKind] IdentList ":" FormalType .
FormalTypeList::= "(" [[pKind] FormalType {"," [pKind] FormalType}] ")"
                                        [":" qualident] .
pKind         ::= VAR | INOUT .
```

2.9 Units und Unitprozeduren

In *Modula-P* gibt es eine weitere Art von Modulen, sogenannte *Units*. Es gibt Definitions- und Implementierungsunits. In einer Unit können Konstanten, Typen und Prozeduren vereinbart werden, allerdings weder Variablen noch lokale Module. Ansonsten gelten die gleichen Regeln wie für `DEFINITION` und `IMPLEMENTATION MODULE` (im folgenden *gewöhnliche Module* genannt). Die in einer `DEFINITION UNIT` definierten Prozeduren werden *Unitprozeduren*, alle anderen *gewöhnliche* Prozeduren genannt. Die einzige Ausnahme bildet das Hauptprogramm (d.h. der Rumpf des Programm `MODULE`s); es wird ebenfalls als Unitprozedur aufgefaßt. Unitprozeduren dürfen nur Wert- oder Inout-Parameter haben. Ansonsten gelten die gleichen Regeln wie für gewöhnliche Prozeduren.
Units dienen dazu, eine andere Bedeutung des Begriffes *Lebensdauer von globalen Variablen* zu definieren. In *Modula-2* gibt es drei *Lebensdauern* von Variablen:

Globale Variable Eine Variable ist *global*, wenn sie in einem Modul deklariert wurde. Sie existiert, solange das Programm abgearbeitet wird. Eine Variable kann eine

Lokale Variable sein, d.h. sie existiert vom Anfang des Aufrufes der sie deklarierenden Prozedur bis zu deren Ende, oder sie ist eine

Haldenvariable , d.h. ihre Lebenszeit wird vom Programmierer selbst *dynamisch* bestimmt. Sie wird auf der Halde durch Aufruf einer Prozedur `ALLOCATE` erzeugt und existiert bis zu ihrer Freigabe durch `DEALLOCATE` (oder vergleichbaren Routinen).

In *Modula-P* wird die Lebenszeit von Variablen, die in Modulen deklariert sind, an die Aufrufe einer Unitprozedur geknüpft. Das heißt also: *alle in gewöhnlichen Modulen deklarierten Variablen, welche von einer Unit (transitiv) importiert werden, beginnen ihre*

Existenz mit dem Aufruf einer Unitprozedur, und ihre Existenz endet mit dem Ende dieser Prozedur. Jede Unitprozedur und jede Instanz (Aufruf) einer Unitprozedur besitzen ihre eigene Instanz dieser globalen Variablen[9].
Die Modulrümpfe aller von einer Unit importierten gewöhnlichen Module werden bei jedem Aufruf einer Unitprozedur - und nur dann - in der üblichen Reihenfolge ausgeführt. Unitmodule haben keine Rümpfe.
Bemerkungen:

- Eine Unitprozedur kann immer auf einem anderen Prozessor mittels der `AT` Anweisung ausgeführt werden. Probleme treten auf, wenn Zeiger auf Haldenvariablen (oder Adressen auf andere Variablen) übergeben werden. Da dies nicht immer zum Zeitpunkt der Übersetzung, noch zur Laufzeit des Programmes erkannt werden kann, ist das Verhalten undefiniert.
- Werden mehrere Unitprozeduren einer Unit (oder mehrerer Units, welche die gleichen gewöhnlichen Module importieren) aufgerufen[10], teilt keine Instanz einer Unitprozedur mit einer anderen die darin deklarierten Variablen.

Die Syntax lautet:

```
CompilationUnit ::= ... | DefUnit | ImpUnit .
DefUnit ::= DEFINITION UNIT ident ";" {import}{unit_defs} END ident ".".
ImpUnit ::= IMPLEMENTATION UNIT ident ";" {import}{unit_decls} END ident ".".
unit_defs  ::= CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
               ProcedureHeading ";" .
unit_decls ::= CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
               ProcedureDeclaration ";" .
```

2.10 Abstrakte Zeit

Der Begriff einer abstrakten Zeit wird im Modul `Time` definiert: `TYPE TIME = INTEGER`. Die Bedeutung der Werte dieses Types wird durch das Laufzeitsystem bestimmt. Werte vom Typ `TIME` können mit den üblichen relationalen Operatoren verglichen werden, wobei `a < b` als „a vor b“ zu lesen ist, etc.

```
DEFINITION MODULE Time;
TYPE TIME  = INTEGER; (* This is a cylic time value.                        *)
TYPE TICKS = INTEGER;
(* from an algebraic view the following should be observed:
 * -:TIME×TIME→TICKS; +:TIME×TICKS→TIME; *,DIV,MOD:TICKS×INTEGER→TICKS *)
CONST sec   = ....;  (* Number of ticks per second, depends on hardware.   *)
CONST cycle = ....;  (* Each cycle seconds, Time() returns the same value.  *)
PROCEDURE time (): TIME;     (* Return the actual time.                     *)
PROCEDURE delay (t : TICKS); (* Suspends the process for t ticks.           *)
END Time.
```

2.11 FOREIGN MODULE

`FOREIGN MODULE` werden benutzt, um von einem *Modula-P* Programm aus Prozeduren aufzurufen, die in anderen Programmiersprachen oder Maschinensprache implementiert sind. `FOREIGN MODULE` haben die gleiche syntaktische Gestalt wie `DEFINITION MODULE`. Lediglich das Schlüsselwort `DEFINITION` wird durch `FOREIGN` ersetzt. Es gibt für `FOREIGN MODULE` damit natürlich kein zugehöriges Implementierungsmodul.
Der Übersetzer kann die Typen der Variablen bzw. der Parameter einer Prozedur einschränken.

[9] Man kann sich das so vorstellen, als ob der Speicherplatz für Variablen, die in Modulen deklariert sind, lokal zu der sie importierenden Unitprozedur allokiert würde.

[10] Z.B. auch in verschiedenen nebenläufigen Prozessen, oder rekursiv

3 Beispielprogramme

3.1 Bedingte kritische Regionen: Ein/Ausgabe

Abb. 1 zeigt, wie mit der `LOCK` Anweisung die Ein/Ausgabe auf dem Bildschirm (oder anderen Ressourcen, die zu einem Zeitpunkt nur von einem Prozeß benutzt werden dürfen) geordnet werden kann. Jeder Prozeß kann hier eine Folge von Ein/Ausgabeoperationen so ausgeben, daß sie nicht von einem anderen gestört werden.

```
MODULE io;
FROM InOut IMPORT
     WriteString,WriteLn,ReadInt;
PROCEDURE square(i:INTEGER;g:GATE);
VAR k : INTEGER;
BEGIN
 FOR k := 1 TO i DO
  LOCK g DO
   WriteString("square");
   WriteInt(i,0); WriteInt(i*i,0);
   WriteLn;
  END;
 END;
END square;
PROCEDURE root(i:INTEGER;g:GATE);
...

VAR io_gate:GATE; i,j:INTEGER;
```

```
BEGIN
 INIT (io_gate);
 PAR
  REPEAT
   LOCK io_gate DO
    WriteString ("squares");ReadInt(i);
   END;
   squares (i, io_gate);
  UNTIL i = 0;
 |
  REPEAT
   LOCK io_gate DO
    WriteString ("roots");ReadInt (j);
   END;
   root (j, io_gate);
  UNTIL j = 0;
 END
END io.
```

Abbildung 1: Beispielprogramm für bedingte kritische Regionen: Ein/Ausgabe

3.2 Das Leser-Schreiber-Problem

Dieses Programmfragment (Abb. 2) zeigt eine Lösung des *Leser-Schreiber-Problemes*: Mehrere Prozesse dürfen lesend auf eine gemeinsame Ressource (Datenstruktur) zugreifen, aber es darf immer nur ein Prozeß diese schreibend verändern.
Die Datenstruktur wird so aufgefasst, als ob sie aus `nr_readers` einzelnen Ressourcen bestünde. Ein Leser kann sich lesenden Zutritt auf genau *eine* dieser Ressourcen verschaffen. Den exklusiven Zugriff auf die ganze Datenstruktur wird dem Schreiber gewährt, indem er alle `nr_readers` Ressourcen benutzt.

```
VAR gate : GATE;
    i : INTEGER;
CONST nr_readers = ...;
PROCEDURE reader (gate:GATE);
BEGIN
 LOOP
  ...;
  LOCK gate DO (* read *) END;
  ...;
 END:
END;
```

```
PROCEDURE writer (gate:GATE);
BEGIN
 LOOP
  ...;
  LOCK gate,nr_readers DO (* write *) END;
  ...;
 END:
END;
BEGIN (* Hauptprogramm *)
 INIT (gate, nr_readers);
 PAR
   [i:1 TO nr_readers] reader (gate)
 | writer (gate)
 END.
END
```

Abbildung 2: Beispielprogramm für bedingte kritische Regionen: Leser-Schreiber

3.3 Client-Server

Das Programmfragment in Abb. 3 zeigt, wie ein einfaches Client-Server-Modell implementiert werden kann. Jeder `server` liest solange Daten und löst dann das Problem, bis er das Kommando `quit` erhält. Jede Instanz der Unitprozedur `server` hat ihren „eigenen“ Keller, der in der gewöhnlichen Prozedur `solve` benutzt wird, um das Problem für die gegebenen Eingabedaten zu lösen. Die Prozedur `new` erzeugt neue Eingabedaten für das Problem, das von den Servern gelöst werden soll, bzw. wenn alle Eingabedatensätze erzeugt wurden, das Terminierungskommando. Es gibt zwei n:m Kanäle: über `job` werden die Daten zu den Servern gesendet, über `res` die Resultate vom Klienten gelesen, im Hauptprogramm werden `max` Serverprozesse gestartet. Sie werden auf alle Prozessoren verteilt. Der Klientenprozeß verteilt zu Beginn an jeden Server einen Datensatz. Danach liest er jeweils ein Resultat, gibt es aus und sendet diesem Server einen weiteren Datensatz. Dauert eine Berechnung zu lange, wird die Prozedur `report_timeout` aufgerufen.

```
DEFINITION UNIT Server;
TYPE tData=RECORD cmd:tCmd;... END;
     tDataChn = CHANNEL OF tData;
PROCEDURE server (in,out:tDataChn);
END Server.
```

```
IMPLEMENTATION UNIT Server;
IMPORT stack;
PROCEDURE solve (VAR data:tData);...
PROCEDURE server (in,out:tDataChn);
VAR data : tData;
BEGIN
 LOOP
  in ? data;
  IF data.cmd=quit THEN EXIT END;
  solve (data); out ! data
 END
END server;
END Server.
```

```
MODULE client_server;
IMPORT Channel, Net, Time;
FROM Server IMPORT tData,...;
VAR job,res:tDataChn;d:tData;
    max, nr_jobs,cur_job,i,j:INTEGER;
```

```
PROCEDURE new (VAR data:tData);
BEGIN
 IF cur_job=nr_jobs THEN data.cmd:=quit
 ELSE INC (cur_job); ...
 END;
END new;
BEGIN (* Hauptprogramm *)
 max:=...;nr_jobs:=max+...;cur_job:=1;
 OPEN(job,Channel.n2m);
 OPEN(res,Channel.n2m);
 PAR
   [i : 1 TO max]
    AT Net.UseAll DO server(job,res)END
 | FOR j:=1 TO max DO new(d);job!d END;
   FOR j := 1 TO nr_jobs+max DO
     ALT
       res?d : print(d);new(d);job!d;
     | WAIT(10*Time.sec):report_timeout
     END
   END;
 END
END client_server.
```

Abbildung 3: Beispielprogramm Client-Server

3.4 Primzahlpipeline

Das *Modula-P* Beispielprogramm in Abb. 4 berechnet Primzahlen[11]. Es zeigt, wie eine beliebig lange „Pipeline“ von Prozessen mittels Rekursion erzeugt werden kann. Man kann sich das so vorstellen, daß es zwei Sorten von Prozessen gibt: einen `generator` und mehrere `worker`. Jeweils ein Kanal verbindet den `generator` mit dem „ersten“ `worker`, bzw. einen `worker` mit dem nächsten. Der `generator` erzeugt die Folge der ungeraden Zahlen, beginnend mit der `3` bis zu der gewünschten Endzahl und sendet diese an seinen direkt verbundenen nachfolgenden `worker` Prozeß. Jeder `worker` Prozeß repräsentiert eine bereits berechnete Primzahl, beginnend mit der Zahl `2`.

[11] Entnommen und angepasst aus: *An Ada Tasking Demo*, Dean W.Gonzalez, Ada letters, Volume VIII, Nr. 5 , Sept,Okt 1988, Seite 87 ff.

Nachdem ein `worker` gestartet wurde, liest er von seinem Eingabekanal die Primzahl, die er repräsentieren soll. Danach liest er die zu testenden Zahlen von diesem Kanal und prüft, ob diese durch seine eigene Primzahl ohne Rest teilbar sind. Ist dies der Fall, wird die Testzahl verworfen, da sie keine Primzahl ist. Andernfalls wird sie an den nächsten `worker` über den Ausgabekanal weitergegeben. Initial werden zwei Prozesse gestartet: ein `generator` und ein `worker`. Jeder `worker` seinerseits startet zwei Prozesse, einen *Prüfprozeß* und den nächsten `worker`, der allerdings wartet, bis ihm eine Primzahl oder das Terminierungssignal (`quit`) geschickt wird.

```
DEFINITION UNIT worker_unit;
TYPE IntChannel = CHANNEL OF INTEGER;
CONST quit = -1;
PROCEDURE worker (in : IntChannel);
END worker_unit.
```

```
MODULE p_prime;
FROM worker_unit IMPORT IntChannel;
FROM InOut IMPORT ReadInt;
FROM worker_unit IMPORT worker,quit;
VAR max,i:INTEGER;out:IntChannel;
BEGIN
 netIO.ReadInt (max);
 OPEN (out);
 PAR
   worker(out) (*startet 1. worker*)
 | (* Testzahlgenerator *)
   out ! 2; (*sendet erste Primzahl*)
   FOR i:=3 TO max BY 2 DO out!i END;
   out ! quit;
 END
END p_prime.
```

```
IMPLEMENTATION UNIT worker_unit;
FROM InOut IMPORT WriteInt;
FROM net IMPORT north;
PROCEDURE worker (in:IntChannel);
VAR out:IntChannel;prime,x:INTEGER;
BEGIN
 in ? prime; OPEN (out);
 IF prime = quit
 THEN RETURN ELSE WriteInt(prime,0)END;
 PAR
   LOOP  (*liest Zahlen*)
    in?x;
    IF x=quit THEN out!quit;EXIT END;
    IF x MOD prime#0 THEN out!x; END;
   END;
 | AT north () DO worker (out) END;
   (*worker auf "noerdl." Prozessor*)
 END;
END worker;
END worker_unit.
```

Abbildung 4: Beispielprogramm Primzahlpipeline

Danksagung

Ich möchte Markus Armbruster, Ullrich Drepper, Helmut Emmelmann, Ralf Hoffart und Hartmut Nebelung für ihre Unterstützung bei der Implementierung und Diskussionen danken.

Literatur

[1] G. Andrews and F. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[2] C.A.R Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice–Hall, Inc., 1985.

[3] INMOS. *The Transputer instruction set - a compiler writers' guide*. Prentice–Hall, 1988.

[4] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia, 1989.*

[5] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming; definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, pages 54–64. IEEE, IEEE Computer Society Press, Los Alamitos, California, April 1992.

[6] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, Heidelberg, New York, third, corrected edition, 1985.